CSE 880

Advanced Database Systems

Database Triggers

S. Pramanik

Introduction

1. Various proposals: IBM's Starburst (research prototype), SQL-3 standard, Oracle Triggers, DB2 triggers, triggers for object-oriented databases.
2. Triggers: Implicitly executed as a result of insert, delete and update transactions on a base table. Both *condition* and *action* are *user* defined.

Potential Applications

1. Allow automatic notification of conditions
(e.g., Student's average grade falling below a threshold)
2. Automatic maintenance of derived attributes.
3. Maintain currency of materialized views.
4. Maintaining mirrored (replicated) databases
5. Others

Oracle Triggers

1. Basic Structure
   (a) Event:
       (Triggering Statement)
   (b) Condition:
       (Trigger Restriction)
   (c) Action:
       (Trigger Action)
2. Example:

```
Table:
 Inventory(Part_no, parts_on_hand, reorder_point)
CREATE OR REPLACE TRIGGER REORDER
  AFTER UPDATE ON Inventory
  FOR EACH ROW
   WHEN(new.parts_on_hand <new.reorder_point)
    BEGIN
     ORDER PARTS
    END
```

   Event: UPDATE ON Inventory, *Condition*: When clause, *Action*: ORDER PARTS (this can be an SQL statement, a stored procedure, etc.).

```
Update Inventory
Set parts_on_hand=parts_on_hand-10
Where Part_no=100
```

3. CHECK versus Trigger:

(a) CHECK checks the data being inserted or updated and is used for data integrity (e.g., prevent user from inserting wrong data) while trigger allows more general user defined actions.

Use table

Employee(EmpNo, EmpName, EmpAddr, EmpSal)

for the following Examples.

Example: Individual employee's salary can be constrained using CHECK inside the definition of Employee as follows:

EmpSal NUMBER(10,2) CONSTRAINT CK_EmpSal CHECK ($EmpSal >$ 1000000.00)

(b) Example of a trigger: constrain on sum of salaries of all employees.

Example:

```
CREATE OR REPLACE TRIGGER MaxTotalSal
  AFTER INSERT OR UPDATE ON Employee
  Begin
   IF 100000000<(select SUM(EmpSal)
        FROM Employee)
   Print "Total Salary Exceeds $100000000.00)
   END
```

12 Oracle Triggers

1. Three Characteristics:
   (a) Action (Event): INSERT, DELETE and UPDATE SQL statement
   (b) Level: Statement level, Row level
   (c) Timing: BEFORE, AFTER
2. Two major difference for Levels:
   (a) How often Trigger fires:
      - Statement level: Trigger fires once per SQL statement
      - Row level: Trigger fires once for each row affected by the SQL statement.

   Example:
   Give 10% raise to all employees
   UPDATE Employee SET EmpSal=1.1 * EmpSal
      - MaxTotalSalary, described on page 5, which is a statement level trigger will fire only once.
      - Following is a row level trigger and will be executed N times where N is the number tuples inserted or updated in Employee table.

```
CREATE OR REPLACE TRIGGER MaxSal
  AFTER INSERT OR UPDATE ON Employee
  FOR EACH ROW
   WHEN (NEW.EmpSal>1000000)
    BEGIN
     Print "Employee Salary exceeds $1000000.00)
    END
```

(b) Visibility:

- Statement level: No accesses to column values currently being updated
  Example: cannot detect individual salary increases.
- Row level: has access to column values currently being updated.
  Example: can detect individual salary increases through NEW.EmpSalary and OLD.EmpSalary
  example: Check more than 50% increase in salary for individual employees

```
CREATE OR REPLACE TRIGGER MaxSal
  AFTER INSERT OR UPDATE ON Employee
  FOR EACH ROW
   WHEN (NEW.EmpSal-OLD.EmpSal>1.5*OLD.EmpSal)
    BEGIN
     Print "Employee Salary increased by by
            more than 50%)
    END
```

3. Timing:

- BEFORE action (i.e. event) occurs
- AFTER action (i.e., event) occurs
- Both BEFORE and AFTER Row level trigger have access to OLD and NEW
  example: Check 50% increase in salary for individual employees. used NEW and OLD.
  Either BEFORE or AFTER row level trigger will do the above job but BEFORE has the opportu-

nity to override the new salary.

- Combination of 12 oracle triggers
- Each of these combinations can be defined on a table more than once.

Column-Name-Clause

Column name clause places tight restrictions on the firing specification. It fires only when the specified columns are affected by the event statements.

Employee(EmpNo, EmpName, EmpAddr, EmpSal)

example:

```
CREATE OR REPLACE TRIGGER MaxSal
  AFTER UPDATE OF EmpSal ON Employee
  FOR EACH ROW
   WHEN (NEW.EmpSal-OLD.EmpSal>1.5*OLD.EmpSal)
    BEGIN
     Print "Employee Salary increased by 50%)
    END
```

```
Assume that OLD EmpSal in the database for EmpNo= 999999 was
(i.e., before the following Update was executed).
Update Employee
Set EmpSal=500000.00
Where EmpNO=999999
```

Will invoke trigger only when Update Statement updates EmpSal.

More Examples

- Maintain a table GoodStudents which has students with grade in a course 4.0. Also indicate the number of rows inserted in each SQL statement.

  StudentCourse(sid, cno, Grade)

```
INSERT  INTO StudentCourse VALUES('A42300', 'CSE880', '4.0
```

Trigger1:

```
CREATE OR REPLACE TRIGGER GoodStudents
  AFTER INSERT ON StudentCourse
   FOR EACH ROW
   WHEN (NEW.Grade=4.0)
      BEGIN
       Insert into GoodStudentsTable
       (NEW.sid, NEW.cno)
       Count=Count+1
      END
```

Trigger2:

```
CREATE TRIGGER InitializeCount
  BEFORE UPDATE ON StudentCourse
   BEGIN
    SET Count=0
   END
```

Trigger3: Can be defined for printing $Count$ by using $AFTER\,UPDATE$ Statement-level trigger.

Similarly trigger has to be implemented for UPDATE and DELETE

Increase the grades of all students in course c1 by .5

```
UPDATE  StudentCourse
SET Grade=Grade+.5
WHERE cno="c1" AND Grade!=4.0
```

```
StudentCourse
sid  cno  grade
s1    c1   3.5
s2    c1   4.0
s3    c1   3.5
s4    c1   3.0
s1    c2   3.5
s5    c2   4.0
```

- Trigger 1 is a row level trigger and will be invoked 3 times.
- Trigger 2 is a statement level trigger and will be invoked only once.

Need a WHEN Clause?

1. WHEN (boolean expression) clause can include any PL/SQL boolean expression.
2. The logic of the boolean test can be moved to the body (action) of the trigger.

Firing Sequence of Multiple Triggers

1. Execute BEFORE statement Trigger
2. LOOP for each ROW affected by SQL statement
   (a) Execute BEFORE ROW Trigger
   (b) Lock and change ROW
   (c) Perform Constraints Checking
   (The lock is not released until the transaction is committed)
   (d) Execute AFTER ROW Trigger.
3. Execute AFTER statement trigger.

Firing Within DELETE CASCADE

1. As each parent is deleted, all children related to the parent are deleted. When child is deleted it's statement and row level triggers are fired.

2. Example:

   Students(sid, name, Dno);
   Department(Dno, Dname)
   ON DELETE CASCADE;

   Delete from Department Where Dno in ('d1', 'd2')

```
All Department BEFORE statement triggers
All  Department BEFORE ROW triggers
    delete Department tuple 'd1'  due to DELETE SQL
ALL Student BEFORE statement triggers due to "Delete Stude
All Student BEFORE ROW triggers due to "Delete Student"
    Delete Student tuple 1
All Student AFTER ROW triggers  due to "Delete Student"
All Student BEFORE ROW triggers due to "Delete Student"
     Delete Student tuple 2
    - - -
    Student AFTER statement
  Department BEFORE ROW
     - - -
  Department AFTER ROW
Department AFTER statement
```

Data Dictionary Views for Triggers

1. View names: $USER\_TRIGGERS$, $USER\_TRIGGERS\_COLS$
2. Attributes:

   $USER\_TRIGGERS$: $TRIGGER\_NAME$, $TRIGGER\_TYPE$ (before statement, etc.), $TRIGGERING\_EVENT$ (Insert, delete, update), $WHEN\_CLAUSE$, etc.

   $USER\_TRIGGERS\_COLS$: $TRIGGER\_OWNER$, $TRIGGER\_NAME$, $TABLE\_NAME$, $COLUMN\_LIST$ (specified in update), etc.
3. Example: Get all trigger names, trigger types, trigger events, column names for updates for table Student-Course.

```
Select TRIGGER_TYPE, TRIGGERING_EVENT, COLUMN_LIST
From USER_TRIGGERS A, USER_TRIGGERS_COLS B
Where  A.TRIGGER_NAME=B.TRIGGER_NAME &
       B.TABLE_NAME="StudentCourse"
```

Cascading of Triggers

The execution of the action part of a trigger may cause the activation of other triggers.
Cascading can be recursive or nested.

1. Recursive trigger:

   When an application updates table T1, which fires trigger TR1 updating table T1. This is a direct recursion because trigger TR1 will be invoked again. Here update comes back to the same table. Recursion can be indirect as well where an application updates table T1, which fires trigger TR1 updating table T2. Trigger TR2 defined on table T2 then updates table T1.

2. Nested triggers:

   If a trigger changes a table on which there is another trigger, the second trigger is then activated and can then call a third trigger, and so on. Here a trigger invokes another trigger.

   Maximum number of cascading is 32 in Oracle;
   When it exceeds 32, all database changes as a result of original SQL are rolled back.

Example:

```
CREATE TRIGGER SalaryControl
  AFTER INSERT, DELETE, UPDATE ON Salary OF EMP
WHEN (Select AVG(Salary) From EMP>100)
THEN Update EMP
     Set Salary=.9*Salary
```

```
        EMP
Employee   Salary
 john       90
 David      100
```

Insert tuple (Rick 200)
How many times the trigger is invoked?

Starburst

1. Research prototype developed at IBM SanJose Research Lab.
2. Based on ECA
   (a) CONDITION: boolean expressed in SQL
   (b) ACTION: SQL statements, rule manipulation statements, transactional instructions such as roll back
3. All are "Statement Level" Triggers invoked *implicitly* or *explicitly*:

   Implicitly: Invoked when the transaction issues a COMMIT- Statement level AFTER trigger

   Explicitly: Using PROCESS RULES
4. Partial ordering of rules
   (System maintains a total order to guarantee repeatability).

Rule Processing Algorithm

1. A rule is marked *triggered* by SQL insert, delete, update statements.
2. A set of triggered rules form the conflict set.
3. Rule Processing Algorithm:

```
1. Select one rule R from the conflict set
   with the highest priority.
2. Evaluate the CONDITION of R
3. If the CONDITION of R is TRUE
   execute ACTION of R.
```

4. Quiscent state: When rule execution terminates and the conflict set is empty.

Transition tables & Visibility

1. Transition tables INSERTED (for INSERT statement), DELETED (for DELETE statement) and OLD-UPDATED for before UPDATE statement) , NEW-UPDATED (for after UPDATE statement) contains tuples from insert, delete and update SQL statements, respectively.

2. All tuples in the transition table correspond to a transaction.

3. Each tuple affected appears at most in one of the transition tables.

4. Each tuple has a net effect of operations.
   Example: insert tuple T1 will put tuple T1 in the INSERTED table. Following update on the same tuple T1, within the same transaction, to T2 will replace T1 by T2 in the INSERTED table.

Examples

Example 1:

If average salary exceeds 100 then reduce each employee's salary by 10%.

```
CREATE RULE SalaryControl ON Emp
WHEN INSERTED, DELETED, UPDATED (Sal)
IF (SELECT AVG(Sal) FROM Emp)>100
THEN UPDATE Emp
     SET Sal=.9*Sal
```

- Assume the following initial values in the Emp table:

Employee	Sal
Stefano	90
Patrick	90
Michael	110

- Insert tuples (Rick, 150) and (John, 120) through a single INSERT SQL statement into the Emp table.

- Initial tuples in the two tables: table initially as follows:

INSERTED		Emp	
Employee	Sal	Employee	Sal
Rick	150	Stefano	90
John	120	Patrick	90
		Michael	110
		Rick	150
		John	120

- The two tables will change as follows as a result of $SalaryControl$ being triggered twice:

INSERTED			INSERTED	
Employee	Sal		Employee	Sal
Rick	135		Rick	121
John	108	=>	John	97

Emp			Emp	
Employee	Sal		Employee	Sal
Stefano	81		Stefano	73
Patrick	81		Patrick	73
Michael	99	=>	Michael	89
Rick	135		Rick	121
John	108		John	97

Here the values in the INSERTED table are not used in the rule but to use the values we have to consider INSERTED as a table and use SQL to access the values.

Example 2:

Add the following Trigger *HighPaid* with lower priority than SalaryControl:
Insert any new tuple into a table *HighPaidEmp* if the *sal* > 100.

```
CREATE RULE HighPaid ON Emp
WHEN INSERTED
IF    EXISTS(SELECT * FROM INSERTED
             WHERE Sal>100)
THEN  INSERT INTO HighPaidEmp
      (SELECT * FROM INSERTED
       WHERE Sal>100)
FOLLOWS SalaryControl
```

- Triggers SalaryControl and HighPaid will be in the conflict set as a result of the insert of the two tuples (Rick, 150) and (John, 120).
- Trigger SalaryControl will be selected and executed first. It will execute recursively twice.

- At this point the INSERTED table is as follows:

INSERTED	
Employee	Sal
Rick	121
John	97

- Trigger *HighPaid* is selected next and executed. and the tuples inserted into the HighPaidEmp table is **"Rick 121"**
- Condition clause can be included in the ACTION clause.
- Base table Emp is used for both the condition and the action clauses of example 1.
- INSERTED table is used for both the condition and the action clauses of example 2.
- Which table is used depends on the function of the trigger. However, temporary tables like INSERTED are smaller and more efficient to use.

IBM's DB2

1. Has similarities to SQL-99 standards
2. DB2 developed based on the experience gained from Starburst and following SQL-99 standards for Triggers
3. As in ORACLE, triggers are activated BEFORE and AFTER the event.
4. BEFORE trigger executes entirely before the event and the database state is before the event.
5. BEFORE trigger cannot modify the database (any tables) from within the trigger by INSERT, DELETE and UPDATE statements.
6. Therefore, BEFORE trigger CANNOT RECURSIVELY activate other triggers.
7. In AFTER trigger, state of the database can be reconstructed to the state before the event (e.g., T MINUS NEW_TABLE UNION OLD_TABLE where T is the target table for the event)
8. As in Oracle it has a row and statement level granularity.
9. For ROW level: Correlation variables NEW and OLD; same as in ORACLE.
10. Syntax of statement level trigger: "FOR EACH STATEMENT": Trigger is executed once for each statement (INSERT, DELETE or UPDATE) in the transaction.

11. Syntax and semantics are different from ORACLE
12. DB2 has transition TABLES for set-oriented events as in Starburst. ORACLE does not have transition tables, it has only correlation variables NEW and OLD for ROW level trigger.
13. Names of the transition tables: INSERTED for INSERT event, DELETED for DELETE event, OLD_UPDATED for before UPDATE event, NEW_UPDATED for after UPDATE event.
14. These tables are for single event while in Starburst tables correspond to multible events within a transaction reflecting the net effect of all the events in the transaction.
15. In DB2 a trigger is defined only on one event, unlike in ORACLE where a trigger can be defined on multiple events, (for example, on INSERT AND UPDATE).
16. Ordering of trigger excutions, when there are multiple triggers invoked on the same event, has some similarities to that of Oracle; first "statement level before" then "row level before," etc. Multiple triggers with the same priority based on the above, are executed in the order in which they were created. For example, the trigger that was created first is executed first; the trigger that was created second is executed second, etc.

Applications (using oracle triggers)

1. Derived data Maintenance:

   Computed Attributes: Attribute value is computed on the fly such as the attribute AverageGrade.

   Materialized Views: Physical up-to-date copy of the view is maintained by the system. Challenge is to keep the physical copy updated as soon as the base tables are updated.

2. Database Replication: Mirrored databases (multiple copies of the same database) are maintained at various locations so that copies are physically close to where it is used to improve response time. Challenge is to keep all these copies in sync.

3. Workflow Management: Organize the working activities within an enterprise. Monitor the assignment of tasks and their coordination.

Materialized Views

Maintaining materialized Views:
Done through

- Refresh approach: Recomputing from *scratch* after each update to base table.
- Incremental approach: Requires computing the positive and the negative deltas, consisting of tuples that should be, respectively, inserted into or deleted from the views.

Example:

```
Student(Sno, Sname);
StudentCourse(cno, sno), sno foreign key
referencing Student(Sno);
     Define view GoodStudents
        SELECT DISTINCT Sno, Sname
        FROM Student, StudentCourse
        WHERE Sno=sno
        GROUP BY sno
        HAVING AVG(Grade)>3.0
```

Events causing recomputation of the view:
insert, delete, update(Grade), update(Sname) for Student and StudentCourse tables.

```
CREATE TRIGGER RefreshGS1
AFTER INSERT, DELETE, UPDATE(Grade) on
 StudentCourse
   DELETE * FROM GoodStudent
   INSERT INTO GoodStudent(Sno, Sname)
     (SELECT DISTINCT y.sno, x.Sname
     FROM  Student x, StudentCourse y
     WHERE Sno=sno
     GROUP BY y.sno
     HAVING AVG(Grade)>3.0)
```

```
Similar Rule for UPDATE on StudentCourse.  We also need
 rules for Student Table.
```

Incremental (more complex):

```
CREATE TRIGGER IncrementalGS1
AFTER INSERT Student
 FOR EACH ROW
  INSERT INTO GoodStudents(Sno, Sname)
    (SELECT DISTINCT Sno, Sname
     FROM Student, StudentCourse
     WHERE sno= new.Sno
     GROUP BY sno
     HAVING AVG(Grade)>3.0)

CREATE TRIGGER IncrementalGS2
AFTER INSERT StudentCourse
  FOR EACH ROW
   INSERT INTO GoodStudents(Sno, Sname)
    (SELECT DISTINCT sno, Sname
     FROM Student, StudentCourse
     WHERE new.sno=Sno
     GROUP BY sno
     HAVING AVG(Grade)>3.0)
```

Similarly, triggers for both delete and update for Student and StudentCourse tables have to be implemented.